计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
63(7): 1803−1815, 2026

本期推荐

深度学习处理器矩阵跨步访存冲突判定方法

张振兴[1,2,3] 赵永威[2] 文渊博[2] 郭 崎[2] 陈云霁[2,4,5]

[1]（中国科学技术大学计算机科学与技术学院 合肥 230026）

[2]（处理器芯片全国重点实验室（中国科学院计算技术研究所） 北京 100190）

[3]（寒武纪科技 北京 100191）

[4]（中国科学院大学 北京 101408）

[5]（中国科学院工业人工智能研究院 南京 211135）

（winkzy@mail.ustc.edu.cn）

A Strided Matrix Memory Access Conflict Determination Mechanism for Deep Learning Processors

Zhang Zhenxing[1,2,3], Zhao Yongwei[2], Wen Yuanbo[2], Guo Qi[2], and Chen Yunji[2,4,5]

[1]（*School of Computer Science and Technology*, *University of Science and Technology of China*, *Hefei* 230026）

[2]（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）

[3]（*Cambricon Technologies*, *Beijing* 100191）

[4]（*University of Chinese Academy of Sciences*, *Beijing* 101408）

[5]（*Institute of AI for Industries*, *Chinese Academy of Sciences*, *Nanjing* 211135）

Abstract With the rapid development of deep learning applications, deep learning processors (DLPs) such as TensorCore GPU, TPU, MLU have risen rapidly and demonstrated excellent performance in matrix operations. To efficiently handle complex matrix data, these processors typically partition sub-matrices and load them into on-chip memory with fixed stride, establishing a distinct pattern of strided memory access. However, devices like NVIDIA H100, Huawei Ascend, and Cambricon MLU currently depend on complex explicit conflict management, lacking dedicated hardware controls for strided memory access conflicts, thus making it challenging to ensure the atomicity and consistency of instruction execution. Traditional memory conflict detection methods focus on scalar operations and have difficulty capturing dependency conflicts among matrix data. To address this issue, we propose a strided matrix memory access conflict determination method that transforms conflict detection into the problem of solving a binary linear Diophantine equation, thereby accurately determining the dependency relationships between memory access instructions. Simulation experiments demonstrate that this approach markedly enhances the memory performance of typical operators in large language models: memory bandwidth utilization can peak at 94% (LPDDR) and 91% (HBM), while the average determination overhead is reduced to 4.68% and 0.31% when compared with the Power test and set intersection methods, respectively. In addition, hardware evaluation based on a 12 nm process

收稿日期：2025-03-01；**修回日期**：2026-03-04

基金项目：国家自然科学基金项目（U22A2028, 61925208, 62222214, 62102398, 62302483）；中国科学院基础与交叉前沿科研先导专项（XDB0660300, XDB0660301, XDB0660302）；中国科学院稳定支持基础研究领域青年团队计划项目（YSBR-029）；中国科学院青年创新促进会和科学探索奖资助项目

This work was supported by the National Natural Science Foundation of China (U22A2028, 61925208, 62222214, 62102398, 62302483), the Strategic Priority Program of Chinese Academy of Sciences (XDB0660300, XDB0660301, XDB0660302), the Project for Young Scientists in Basic Research of Chinese Academy of Sciences (YSBR-029), and the Youth Innovation Promotion Association of Chinese Academy of Sciences, and Xplore Prize.

通信作者：郭崎（guoqi@ict.ac.cn）

shows that the determination design occupies only 0.023 33 mm^2 and the power consumption is 4.119 4 mW. Overall, this mechanism transitions from complex explicit management to efficient hardware-based determination, effectively improving the memory management of DLPs.

Key words AI computing systems; deep learning processor; strided memory access; memory access conflict; linear Diophantine equation

摘 要 随着深度学习应用的迅猛发展, TensorCore GPU, TPU, MLU 等深度学习处理器(deep learning processor, DLP)迅速兴起,并在矩阵运算中展现出优异性能。为高效处理复杂矩阵数据,这类处理器通常以固定步长将子矩阵分块加载到片上存储器,形成特定的跨步访存(strided memory access)。然而,英伟达 H100、华为 Ascend 及寒武纪 MLU 等设备目前均依赖复杂的显式冲突管理,缺乏对矩阵跨步访存冲突的硬件调控,致使指令执行原子性和一致性难以保障。传统访存冲突检测方法侧重于标量运算,难以捕捉矩阵数据间的依赖冲突。针对这一现状,提出了一种矩阵跨步访存冲突判定方法,将冲突检测归约为二元一次不定方程求解,从而精确判断访存指令间的依赖关系。模拟实验表明该方法显著优化了大模型典型算子的访存性能:访存带宽利用率最高可达 94%(LPDDR 存储器)与 91%(HBM 存储器),且与 Power 测试和集合相交方法相比,平均判定开销分别降至 4.68% 和 0.31%。此外,基于 12 nm 工艺的硬件评估显示,比较器面积仅 0.023 33 mm^2,功耗 4.119 4 mW。总体而言,该机制由复杂的显式管理转向高效硬件判定,有效提升了深度学习处理器的存储管理能力。

关键词 智能计算系统;深度学习处理器;跨步访存;访存冲突;不定方程

中图法分类号 TP332

DOI: 10.7544/issn1000-1239.202550124 **CSTR:** 32373.14.issn1000-1239.202550124

近年来,随着人工智能和深度学习技术的兴起,深度学习处理器(deep learning processor, DLP)以及智能计算系统已成为计算体系结构领域的重要研究方向[1-2]。DLP 是一种面向深度学习任务优化设计的专用硬件,区别于以标量或向量为基本数据单元的传统处理器,DLP 采用矩阵作为数据处理的基本粒度[3-4]。其芯片配置矩阵运算单元和矩阵存储单元,专门用于直接处理矩阵数据,从而显著提升算力和能效。因此, DLP 能高效执行矩阵运算、卷积等计算任务。

DLP 使用矩阵访存指令存取数据,其特点为跨步访问。例如,在 4×4 矩阵乘法中,运算单元需分批加载 4×4 子矩阵到片上存储器,再将结果写回片外存储器。图 1 显示 1 个 4×8 矩阵的线性存储方式,其中深色区域代表 1 条访存指令的操作数,这些数据分散存于多个子数据段,每段间隔固定步长。传统访存模式难以高效访问此类数据。相比之下, DLP 专为跨步读取设计的访存指令能降低译码和寻址开销,有效读取分散数据,提升访存带宽,并适用于卷积、特征图存取与矩阵运算。理想情况下,矩阵访存操作应具备原子性,即操作不可分割,要么完整加载或存储整个相应的数据块,要么完全不执行,特别是在异步并发执行访存指令时,这是保证计算正确性的关键。

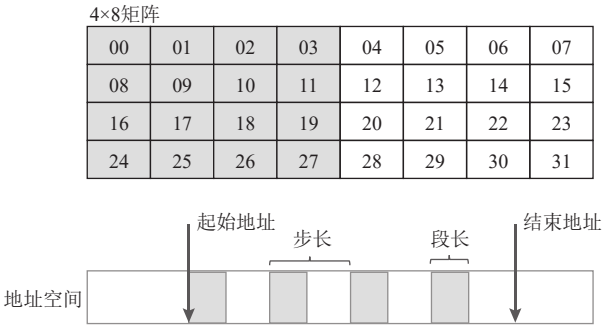


Fig. 1 Strided pattern in matrix memory access operations

图 1 矩阵访存操作中的跨步模式

然而, DLP 的访存控制设计仍存在一个关键性问题,即如何判定矩阵访存指令是否存在冲突,这对于 DLP 生态建设至关重要。早期的深度学习计算系统主要依赖高性能库进行计算任务优化,用户通常调用经过精心调优的库函数,而不直接涉及底层指令,因此无需考虑访存冲突问题。然而,随着深度学习计算朝着开放生态发展,用户可以通过智能编程语言(如寒武纪 BANG C[5] 编程语言、Triton[6])和深度学习编译器[7],更直接地控制底层访存和计算行为。这种增强的灵活性在提升可编程性和优化潜力的同时,也显著增加了因访存指令参数配置不当而引发冲突的风险。

事实上，当前主流的DLP平台，包括英伟达Hopper GPU、华为Ascend昇腾处理器和寒武纪MLU思元处理器等，在硬件层面普遍缺乏对矩阵访存冲突的自动检测机制，这使得冲突规避的责任主要由开发者或上层软件栈承担。这种现状带来了显著风险：由于缺乏硬件层面的冲突判断能力，处理器在并发执行访存指令时，用户不当的参数配置以及同步设置极易引发未定义行为，这些行为包括但不限于数据竞争、数据覆盖、访存队列阻塞，不仅直接威胁计算结果的准确性，还可能通过链式传播影响计算系统稳定性。此外，不同硬件世代产品可能对冲突场景采取不同的处理策略（如覆盖冲突数据或者抛出硬件异常）导致同一程序在不同平台上的行为不一致，从而削弱程序的可移植性和深度学习计算生态的可靠性。因此，构建一个具有高效冲突检测机制的访存管理机制，为DLP访存指令明确一个清晰的编程界面，已成为DLP架构设计中的关键需求。

针对矩阵跨步访存的冲突判定，需要一种精确且高效的检测机制。传统的访存管理技术（如寄存器重命名和乱序执行）主要适用于标量访存场景，在矩阵计算中难以应对复杂的跨步访存模式。矩阵访存包含若干个地址，地址冲突结合指令的读写方向就可以判断是否存在读后写（WAR）、写后读（RAW）或写后写（WAW）等数据冲突行为。集合相交法通过循环展开直接对比子矩阵元素的访存地址，虽然能够精确检测地址冲突，但其计算复杂度较高，增加了系统开销。而区间覆盖法通过将子矩阵的起始地址和结束地址视为连续区间，判断区间是否存在交集，从而检测地址冲突，该方法与Banerjee-Wolfe测试[8-9]等价。然而，该方法忽略了矩阵条带之间的“空隙”，可能导致误判。因此，集合相交法和区间覆盖法在性能和精度上均存在一定局限性。

为解决上述问题，本文提出了一种基于不定方程求解的矩阵访存冲突检测方法。我们将访存冲突判定建模为二元一次不定方程是否有解的问题，精确判断访存指令是否存在冲突。实验结果表明，相较于区间覆盖法，本文方法有效避免了粗粒度检测导致的误判，访存带宽利用率显著提升。LPDDR存储器的带宽利用率平均值从72%提升至89%，HBM存储器的带宽利用率平均值从32%提升至88%，访存带宽利用率最高分别可达94%和91%。同时，与Power测试和集合相交法相比，本文方法的冲突判定延迟降低了1~2个数量级，显著提高了计算效率。同时基于12 nm工艺综合结果显示，面积仅0.023 33 mm^2，功耗仅4.119 4 mW。

本文的主要贡献如下：

1）在DLP架构设计上实现了矩阵跨步访存的冲突判定机制，确保访存指令的原子性，避免了未定义行为带来的不可预测性；

2）提出了跨步访存冲突的形式化分析方法，将冲突判定建模为不定方程是否存在合法解，判定求解的时间复杂度大幅降低；

3）在模拟仿真平台评估方法的有效性，实验表明：相较于Banerjee–Wolfe测试，本文方法下的访存带宽利用率最高可提升至94%（LPDDR）和91%（HBM），本文方法的判定开销相较于Power测试和集合相交法，降低至4.68%和0.31%，低了1~2个数量级。同时仅需很小的硬件开销。

1 背　　景

1.1 深度学习计算中的张量数据存储

张量作为深度学习框架中描述多维数据的核心数据结构，承载了网络的输入、参数、激活值及梯度信息。张量本质上是高维数组，其在内存中以线性排列，但其多维结构反映了数据之间的内在关系。大语言模型网络中，常见的数据存储格式包括BSH（批量、序列长度、隐层大小）、BNSD（批量、多头数、序列长度、隐层单元尺寸）和BSND（批量、序列长度、多头数、隐层单元尺寸）。卷积神经网络中，常见的数据存储格式包括NCHW（批量、通道、高度、宽度）和NHWC（批量、高度、宽度、通道）。其中，不同的排列方式直接决定了各维度数据的连续性。例如，NCHW格式使得高度与宽度维度上数据连续，利于在空间上进行连续访存；而NHWC格式则在通道维度上保证连续，更利于多通道之间求和。不同的存储格式选择会直接影响数据访问的局部性和整体访存效率。

在实际运算时，深度学习计算通常采用数据分块（tiling）技术[10]，将大规模多维数组划分为更小的子块，将数据加载到片上存储器后由矩阵运算器处理。这样既能提升片上存储器的利用率，也能降低片外存储器的访问开销，同时避免容量失效、访问放大、临界步幅和伪共享等问题[11]。例如，在卷积计算中，可以通过空间维度（高度、宽度）或通道维度进行分块，如图2所示，输出计算先按照输出通道维度拆分，输入特征经过运算得到一部分输出通道的结果。同时再将输入数据按照空间维度拆分成2个子任务

独立执行，共同完成输出通道维度的部分计算。然而，这种数据分块策略，特别是在跨多个张量维度进行计算时，会导致数据访问呈现矩阵跨步访问。例如图中的输出特征，其数据访问地址并非完全连续，而是由多个具有固定步长(stride)和段长(segment length)的数据段组成。

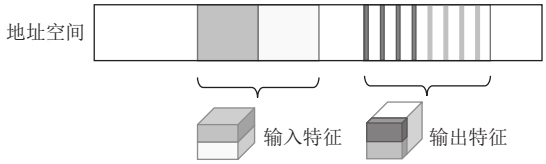


Fig. 2 DLP processes the data blocks

图 2 DLP 对数据分块处理

1.2 DLP 中的跨步访存支持

DLP 通常采用如图 3 所示的专用访存架构，以支持矩阵计算的高效数据流。典型的处理器设计中会集成矩阵运算单元、向量运算单元和标量运算单元等多种计算核心，以适应不同类型的计算任务。为了避免传统缓存层次结构带来的性能瓶颈，DLP 普遍采用便笺式存储器(scratchpad memory, SPM)进行显式的数据管理。这里的便笺存储器指的是由软件进行管理的片上 SRAM 存储单元，包括矩阵存储单元、向量存储单元和标量存储单元。片外与片上的数据搬运主要通过直接访存(direct memory access, DMA)模块完成，该模块支持大粒度的矩阵装载和存储指令，能够直接在主存储器和片上存储单元之间高效传输数据块。这种设计减少了对大量细粒度标量 load/store 指令的依赖，简化了指令流水线，并降低了与取指和译码相关的开销。在典型的张量处理流程中，输入数据和权重首先通过直接访存模块从片外存储器加载到片上存储器，计算单元处理得到中间结果或最终结果后再写回片外存储器。这一过程中，数据的搬运往往涉及复杂的跨步访存模式。

DLP 基本支持跨步访存，其中寒武纪 MLU 是当前对跨步访存支持最为充分的硬件架构之一，其设

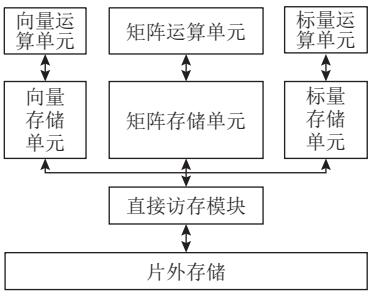


Fig. 3 Processing unit and SPM design of DLP

图 3 DLP 运算单元与便笺式存储器设计

计充分体现了对智能计算任务高性能需求的深刻理解。以寒武纪 MLU 为例，其 BANG C 编程语言提供了参数化的 2 维内存拷贝接口(即函数 *__memcpy*)[12]，用户可通过参数化定义灵活设置内存拷贝模式(如地址、步长、段长和分段数)。同时，函数 *__memcpy* 支持在神经元存储(NRAM)、权值存储(WRAM)、共享存储(shared RAM)以及全局存储(global DRAM)之间进行数据传输，从而充分发挥多级存储层次的性能优势。华为 Ascend 系列加速卡亦采用基于矩阵粒度的跨步访存指令[13]，而这些指令均通过 DMA 模块完成数据搬运，只有部分标量加载操作可绕过 DMA 模块。英伟达 Hopper 架构针对矩阵访存引入了 TMA (tensor memory accelerator)[14] 机制，该机制自动处理地址生成以及步长计算，实现了片外存储和片上存储间的异步数据传输，优化了深度学习模型的训练与推理性能。这些先进的访存机制极大地提升了非连续数据的交互效率，但同时也引入了新的编程与管理挑战，特别是它们通常缺乏内建的矩阵冲突检测与处理机制。

1.3 研究动机

开放的软件生态需要硬件提供强大的管理机制来适应各种复杂的情况。DLP 通过固定大小的硬件，支持任意大小的计算负载，数据分块计算时不同指令会对同一个多维数组精细化切分，因此访存冲突难以规避。异步访存指令在多通道存储器配置下既可能并发执行，也可能因访问相同存储地址引发冲突，如读后写、写后读或写后写冲突。这些冲突不仅影响计算正确性，还可能造成数据覆盖、未定义行为，甚至危及整个计算生态的稳定性。过去，深度学习计算依赖优化库函数规避访存冲突，而随着 DLP 生态的开放，开发者对底层访存指令的直接控制使访存行为更加复杂化。

目前，尽管现代处理器提供了强大的矩阵跨步访存指令，但硬件层面的冲突检测与解决机制尚不完善。这一现状要求开发者承担显式管理并发访问的责任，如图 4 所示。以寒武纪 MLU 的 BANG C 编程为例，开发者连续使用 2 条异步拷贝函数 *__memcpy_async*，并且因为地址重叠产生了写后写冲突。为了解决冲突，开发者应合理拆分数据，并在适当位置执行 *__sync_io* 和 *__sync_compute* 等同步操作；否则，可能因访存指令冲突而产生未定义行为。华为 Ascend 处理器的编程指南明确指出，对于目标内存区域存在重叠的并发 DataCopy 操作，必须插入同步屏障(如 PipeBarrier)以强制串行化，否则可能导致“异常数

据”。此外，在针对英伟达 Hopper 平台更为复杂的 TMA 代码测试中，矩阵访存场景也存在并发冲突问题，证实其 *cp.async.bulk* 操作存在数据不一致风险。所以在现阶段，主流 DLP 尚未实现对矩阵访存冲突的自动检测与防护机制。因此，为确保矩阵访存操作的原子性、跨平台执行行为的一致性以及整个计算生态的健壮性，研究并设计一种针对矩阵访存冲突的自动检测和判定方法，成为当前高性能计算体系结构领域一项紧迫且关键的任务。

DLP 中的矩阵访存通常以单条指令形式操作大块连续或分段的内存区域，这种设计简化了访存指令集，但也对硬件和软件提出了更高要求。这类矩阵访存的操作粒度是动态变化的，其内存布局由多个参数灵活配置，导致具体执行时间亦随配置和系统状态动态变化，极大地增加了访存行为的复杂性。在开发者拥有完全控制权的情况下，数据的存储布局拥有更高自由度，但也可能导致多个数据段间复杂的交错和重叠模式。与标量访存不同，大粒度的矩阵访存冲突涉及多维数据的复杂比对，不再是单个元素的直接比较。传统寄存器重命名、乱序执行等访存管理方案主要针对标量计算，在矩阵计算场景下难以有效应对跨步数据的访问模式。这种模式通常包含非连续的多段数据访问，涉及复杂的空间布局变化，因此需要设计全新的冲突判定机制。

```
// 异步拷贝指令目的区域重叠，产生写后写冲突
// 寒武纪MLU：
__memcpy_async(dst, src, …);
__memcpy_async(dst+ 32, src, …);


// 华为Ascend：
AscendC::DataCopy(dst_0, src_0, dstSliceInfo_0…);
AscendC::DataCopy(dst_1, src_1, dstSliceInfo_1…);


// 英伟达：
ptx::cp_async_bulk_tensor(…);
ptx::cp_async_bulk_tensor(…);
```

(a) 产生访存冲突

```
// 解决冲突
// 寒武纪MLU：
__memcpy_async(dst, src, …);
__sync_io();
__memcpy_async(dst+ 32, src, …);

// 华为Ascend：
AscendC::DataCopy(dst_0, src_0, dstSliceInfo_0…);
AscendC::PipeBarrier<PIPE_V>();
AscendC::DataCopy(dst_1, src_1, dstSliceInfo_1…);

// 英伟达：
ptx::cp_async_bulk_tensor(…);
ptx::cp_async_bulk_commit_group();
ptx::cp_async_bulk_wait_group(ptx::n32_t<0>{});
ptx::cp_async_bulk_tensor(…);
```

(b) 显式解决冲突

Fig. 4 Explicit memory access conflict management in DLP programming

图 4 DLP 程序设计中的显式访存冲突管理

矩阵跨步访存的冲突判定要求精确且高效的检测机制。如图 5 所示，矩阵切分后的相邻子矩阵 ***A*** 和 ***B*** 在存储中的数据布局是交错的，但是地址范围并未出现局部重叠。判断 2 个子矩阵是否存在冲突（数据范围重叠且包含依赖关系）可通过集合交集实现：将子矩阵地址展开成集合，通过查找重复元素来判断数据范围重叠及依赖关系。这种“集合相交法”能够精确描述冲突，但因元素数量急剧增加而导致查找对比开销巨大。对于连续数据，基于编译原理中静态分析的 Banerjee-Wolfe 测试[8-9] 方法，利用矩阵的首尾地址（上下界）形成最小连续区间，直接判断区间重叠，避免逐元素对比。例如图中子矩阵 ***B*** 和 ***C*** 的最小包含连续区间无重叠，可直接判断无地址冲突。然而，基于上下界区间有无重叠的判断方法在数据交错时可能误判冲突，例如子矩阵 ***A*** 和 ***B*** 的情况。总体而言，“集合相交法”精确但效率低下，“区间覆盖法”高效但不够精确。为应对矩阵数据的复杂访存模式，需要设计更加精细化且低开销的冲突检测方法，以确保访存指令的原子性和计算生态的稳定性，同时提高计算效率与正确性。

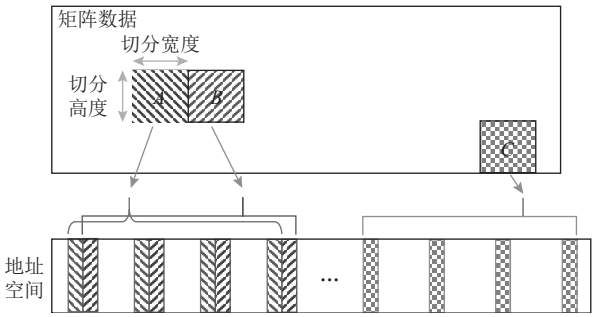


Fig. 5 The positional and address relationships between submatrices

图 5 子矩阵之间的位置与地址关系

2 跨步访存冲突判定的形式化分析

2.1 跨步访存地址冲突的形式化定义

在片外存储环境中，数据的读写可视作对若干

地址集合的访问过程。为探讨当这些集合在存储空间中出现重叠时所引发的“冲突”问题,本节先给出如下形式化定义,并阐述其在离散与连续场景中的适用性。

定义 1. 地址冲突。设 S_1 与 S_2 分别表示 2 条指令所访问的片外存储地址集合。若满足

$$S_1 \cap S_2 \neq \varnothing \quad \Leftrightarrow \quad \exists x \in S_1 \text{ and } x \in S_2,$$

则称这 2 条指令之间存在地址冲突。

定义 1 并未要求 S_1 与 S_2 必须为连续区间或离散子集,只要在存储空间中存在交集便可判定冲突。若需更细粒度地分析冲突检测的复杂度,通常会对地址区间进行结构化描述。当访存地址是连续区间时,v代表起始地址,δ代表偏移量,并且l代表区间长度,则S_1与S_2可记作:

$$S_1 = \{v_1 + \delta \mid \delta \in [0, l_1)\},$$
$$S_2 = \{v_2 + \delta \mid \delta \in [0, l_2)\}。$$

此时,2 个集合的交集非空可表示为

$$\exists \delta_1 \in [0, l_1),\ \delta_2 \in [0, l_2),\ \text{使得}\ v_1 + \delta_1 = v_2 + \delta_2。$$

此处给出了地址冲突发生的充要条件:只有当 2 条指令访问的地址范围在数轴上出现重叠时,才会产生冲突。对于连续区间,可通过区间端点的比较来快速完成冲突检测,若S_1 的右端点大于 S_2 的左端点,并且S_2 的右端点大于 S_1 的左端点,可推得区间交集为非空。

2.2 跨步访存地址冲突判定

当访存操作呈跨步状或更复杂的离散模式时,将其视作 1 个区间并单纯依赖区间端点可能产生过度保守的判定。如 2.1 节所述,区间覆盖法仅依据 $\min\{S_1\}, \max\{S_1\}, \min\{S_2\}, \max\{S_2\}$ 这类上下界信息来对区间进行判断,很可能把部分实则不相交的子集错误地视为冲突,导致判定准确率下降。为此,需要借助更精细的数论工具(如不定方程、最大公约数检验等),对可能出现的地址重叠进行严格刻画,从而在理论上给出更准确的冲突判定准则。为更严谨地刻画跨步访存下的地址分布,先给出定义 2。

定义 2. 跨步访存的地址集合。令v为基准起始地址,s为分段之间的步长,m为分段数量,l为段长(上述参数均由指令提供)。设 x 为分段索引,p 为该分段内的偏移量,则该矩阵跨步访存指令所访问的地址集合可记为

$$S = \{v + s \times x + p \mid x \in [0, m),\ p \in [0, l)\}。$$

在定义 2 下,若有 2 条指令分别对应集合 S_1与S_2,则S_1与S_2可记作:

$$S_1 = \{v_1 + s_1 \times x + p \mid x \in [0, m_1),\ p \in [0, l_1)\},$$

$$S_2 = \{v_2 + s_2 \times y + q \mid y \in [0, m_2),\ q \in [0, l_2)\}。$$

此时其地址冲突可通过检查以下条件来判定:

$$\exists x \in [0, m_1),\ y \in [0, m_2),\ \text{使得}\ s_1 \times x - s_2 \times y = C,$$

即上述二元一次不定方程由是否存在合法整数解来判定,其中

$$C \in S_C,\ S_C = \{v_2 - v_1 + (q - p) \mid p \in [0, l_1),\ q \in [0, l_2)\}。$$

为了更清晰地分析冲突判定的复杂性,以下将分别探讨 2 种情形:段长为 1 的简单情形和段长大于 1 的复杂情形。

2.2.1 段长为 1 的情形

在最简化情形下,若段长均等于 1,则 $p = q = 0$,上述判定可归结为求解不定方程

$$ax + by = c,$$

其中 $a = s_1$, $b = -s_2$, $c = v_2 - v_1$。根据数论理论,方程 $ax + by = c$ 存在整数解的充要条件是 c 为 a 与 b 的最大公约数 $\gcd(a,b)$ 的倍数。由此可知,若 c 不是 $\gcd(a,b)$的倍数,则方程无解,意味着必然不发生冲突;若 c 是 $\gcd(a,b)$ 的倍数,则方程有解,意味着可能存在冲突,需要进一步验证该解在 $x \in [0, m_1)$和$y \in [0, m_2)$ 区间内是否可行。

二元一次不定方程 $ax + by = c$ 的具体解可借助扩展欧几里得算法求得,求解开销低,时间复杂度为 $O\left(\log(\min(a,b))\right)$。算法首先给出该方程的 1 个特解 (x_0, y_0),再利用通解形式:

$$x = x_0 + \frac{b}{d}k,\ y = y_0 - \frac{a}{d}k,\ k \in \mathbb{Z},\ d = \gcd(a,b)。$$

判断其是否落在约束范围内,从而决定地址冲突是否真实发生。为方便理论推导,经常考虑以下 3 个方程组:

1)基本方程$ax + by = c$;

2)归一化方程$a'x + b'y = c'$(其中 $a' = a/d$, $b' = b/d$, $c' = c/d$);

3)标准型方程$a'x + b'y = 1$。

基本方程和归一化方程仅在系数上相差 1 个常数因子 d,可通过除以 d 直接将基本方程转化为归一化方程,因此解相同;标准型方程若存在特解 (x_0, y_0),则利用c'的比例放缩可得到归一化方程的通解,从而也可推出基本方程的通解。该推导过程充分展现了三者间的内在联系,并为跨步访存冲突判定提供了严格的理论基础。

2.2.2 段长大于 1 的情形

若段长大于 1,则集合

$$S_C=\{(v_2-v_1)+(q-p)|p\in[0,l_1),q\in[0,l_2)\}。$$

需进一步考虑 p 与 q 的多种取值所构成的区间。当 S_C 中的某些值不能被 $\gcd(a,b)$ 整除时，对应方程必然无解；只需对能够被 $\gcd(a,b)$ 整除的子集进行求解：

$$S'_C=\{x\in S_C|x\equiv 0(\bmod d)\},\ d=\gcd(a,b)。$$

由于在实际访存场景下 p, q 均取有限范围，故 S'_C 中可行解的个数也是有限的。该子集中每一个数均可表示为 $d\times t$（$t\in\mathbb{Z}$，且 t 落在相应区间），因此均可回到不定方程 $ax+by=d\times t$ 的通解形式进行检验；若存在满足 $x\in[0,m_1)$ 且 $y\in[0,m_2)$ 的整数解，则说明访存指令之间确有地址冲突。值得注意的是，段长大于 1 虽然在集合层面增添了更多的重叠检测可能性，但因其仍可分解为离散的整数偏移量（即 $d\times t$），并通过同样的数论方法求解，最大公约数和标准型方程的特解只求了 1 次，然后重复利用该信息乘以不同的 t 即可得到归一化方程和基本方程的特解，故并未实质性增大冲突判定的复杂度。

综上所述，基于不定方程与最大公约数的分析方法，为跨步访存冲突检测提供了完备且严谨的理论框架。该框架可进一步与具体的 DLP 硬件设计或编译器优化策略相结合，用于减少访存冲突导致的指令阻塞，提升系统整体性能和并行执行效率。

3 跨步访存冲突判定机制及实现

基于前述跨步访存冲突的挑战和冲突判定的数学模型，本节提出一种访存冲突动态判定机制，并详细其设计与实现细节。该机制的核心思想在于：充分利用跨步访存的地址特征和访存指令的元信息（起始地址、间隔步长、段数目、段长等），结合不定方程求解，快速、准确地判断访存指令之间是否存在地址重叠。为了实现这一目标，我们设计了一种基于等待队列和执行队列分离的调度策略，并配置了专用的冲突比较器模块（基于硬件状态机实现，进行运行时动态检测）进行冲突解决，以确保访存指令的正确性和并发性。

DLP 通常采用直接访存模块以支持大粒度访存指令，因为指令可以操作矩阵数据，数据在存储器中并不一定连续排布，其离散性和多样性大幅增加了冲突检测的复杂性。为了在这种情况下依然能够保证访存操作的正确性，并避免数据竞争和一致性问题，具有数据依赖的指令之间仍然必须保持顺序执行。此外，DLP 中额外添加了同步指令（sync），来显式地指导指令流水线是应该继续发射、并行执行，还是需要等待之前的指令完成以保证满足依赖关系。依赖关系则通过判断 2 条指令所访问的地址是否交叠，以及是否存在读后写、写后读以及写后写冲突决定。DLP 不采用复杂的乱序执行流水线。通过设置指令队列以实现计算指令和访存指令的并发，同时通过数据依赖判定来避免子矩阵数据冲突。如图 6 所示，指令缓存中的指令序列经过译码后，计算指令与访存指令分别被分配到 2 个独立的队列，以供顺序执行。在执行阶段，计算指令由运算单元顺序执行。访存指令经过地址生成单元处理，利用子矩阵跨步特性展开地址生成 DMA 微码，然后由 DMA 访存模块执行。若遇到同步指令，将不加入访存队列，直至所有已发射指令执行完成后再恢复后续指令的发射。需要强调的是，虽然访存指令在处理器流水线中是顺序发射的，但由于缓存命中情况和片外存储访问延迟的不确定性，这些指令在实际执行过程中可能会乱序完成。

通常 DLP 的片外存储包括多个存储通道，为充分利用存储通道之间的并发性，从而进一步提升访存性能，访存队列被细化为等待队列（Q_W）和执行队

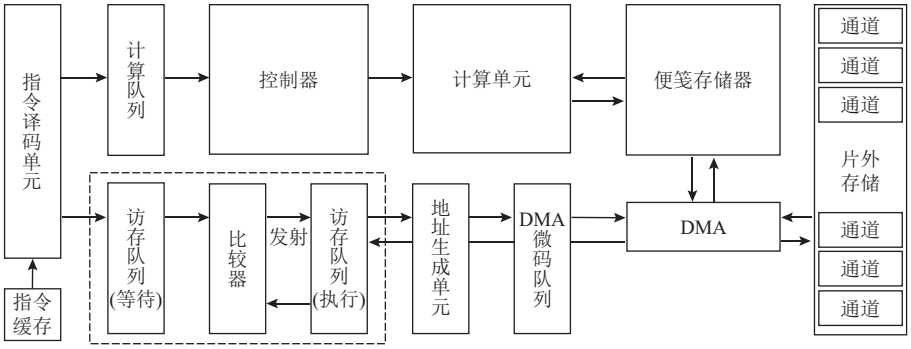


Fig. 6 The overview of architecture

图 6 结构概览

列(Q_R)。如图 6 所示，等待队列按照先进先出的原则调度指令，引入冲突比较器以高效检测并解决访存指令间的潜在冲突，而执行队列已经通过检测机制确保无冲突，从而确保并发执行的正确性。冲突比较器能够动态判断访存指令之间的潜在冲突，具体来说，比较器通过分析访存指令的地址范围、访问类型(读或写)和依赖关系，快速判定是否存在读后写、写后读或写后写冲突。因此，我们通过细化访存队列结构，引入基于冲突比较器的动态检测机制，可以实时监测访存指令之间的地址依赖性，优化内存访问的并发性，从而提高访存操作的整体效率。

算法 1 描述了访存队列的具体处理流程。首先，等待队列Q_W按照先进先出的顺序调度访存指令，并取出队首指令。接着，函数 *HasConflict* 将该指令与执行队列Q_R中的所有指令进行比较，以判断是否存在冲突，该过程依赖于基本的函数 *Compare*，该函数基于访存地址、访问类型及数据依赖关系执行指令间的冲突检测。然后，如果检测到冲突，调度器将暂停调度该指令，直至相关冲突指令全部执行完毕。若无冲突但执行队列已满，则继续等待，直至执行队列有空余位置。最后，当指令既无冲突且执行队列中的指令数量低于设定阈值时，该指令被推入执行队列，进入执行阶段。此外，当队列中首位置的指令因冲突被阻塞时，可以检查其与紧随其后的指令之间是否存在冲突；若不存在冲突，则交换它们的顺序，继续判断，从而实现一个预取深度为 1 的预判。

算法 1. 访存队列处理与冲突检测流程。

输入：等待队列 Q_W，执行队列 Q_R，执行队列容量 N；

输出：访存队列优化完成。

① function *HasConflict*(*current_inst*, Q_R)
② for each q in Q_R
③ if *Compare*(*current_inst*, q)
④ return true; /*检测到冲突*/
⑤ end if
⑥ end for
⑦ return false; /*无冲突*/
⑧ end function
⑨ /*初始化 Q_W, Q_R, N*/
⑩ while not Q_W.*empty*() do
⑪ *current_inst* ← Q_W.*top*();
⑫ if *HasConflict*(*current_inst*, Q_R) then
⑬ continue; /*等待冲突指令完成*/
⑭ end if
⑮ if $|Q_R| \geqslant N$ then
⑯ continue; /*等待执行队列有空位*/
⑰ end if
⑱ *current_inst* ← Q_W.*pop*();
⑲ Q_R.*push*(*current_inst*);
⑳ end while

上述设计的核心在于利用冲突比较器对访存指令实施动态管理，这同时确保访存指令操作的原子性。当检测到访存地址范围存在交集时并且存在依赖关系，等待队列会暂停调度该指令，直到冲突指令完全执行完毕后才继续发射。这保证了指令的执行过程不会被其他并发操作打断或干扰，使操作要么完整执行，要么完全不执行，从而达到原子性的要求。若无冲突，系统还会检查执行队列的容量限制，确保硬件资源合理使用。具体而言，函数 *HasConflict* 采用短路运算优化冲突检测流程，当检测到首个冲突后立即返回结果，避免冗余比较操作。

算法 2 描述了用于检测 2 条访存指令是否存在冲突的函数 *Compare*。该函数通过对指令元信息的解析与分析，判断 2 条指令是否会在访存过程中发生冲突，具体流程如下：首先获取输入指令的访存操作类型(读或写)，若 2 条指令均为读操作，因为读操作之间不会产生冲突，直接返回无冲突即可。然后解析指令元信息，得到 2 条指令对于片外访存的起始访存地址、地址递增模式、每次访存操作的数据长度和跨步访存的分段次数等。最后判断 2 条指令在访存范围上的最大可能区间是否存在交集，若区间无交集，则 2 条指令一定不会发生冲突，这样可以避免后续的运算；若区间存在交集，则通过 2.2 节中的数学方法进一步判断冲突是否存在地址上的交集。

算法 2. 比较函数 *Compare* 运算过程。

输入：指令 *inst_a*，指令 *inst_b*；

输出：是否冲突(布尔值)。

① *rw_a* ← *get_direction*(*inst_a*);
② *rw_b* ← *get_direction*(*inst_b*);
③ if *rw_a* = *kRead* and *rw_b* = *kRead* then
④ return false; /*均为读操作，无冲突*/
⑤ end if
⑥ *v_*1, *s_*1, *l_*1, *m_*1←*meta_info*(*inst_a*); /*解析 *inst_a* 的元信息*/
⑦ *v_*2, *s_*2, *l_*2, *m_*2←*meta_info*(*inst_b*); /*解析 *inst_b* 的元信息*/
⑧ if *MaxRangeIntersection*(*v_*1, *s_*1, *l_*1, *m_*1, *v_*2,

*s*_2, *l*_2, *m*_2) = false then
⑨ return false; /*最大范围区间无交集*/
⑩ end if
⑪ if *SolveEquation*(*v*_1, *s*_1, *l*_1, *m*_1, *v*_2, *s*_2, *l*_2, *m*_2) = true then
⑫ return true; /*方程存在解，存在冲突*/
⑬ end if
⑭ return false。 /*无冲突*/

综上所述，本文通过细化访存队列结构和引入动态冲突检测机制，完善了 DLP 的访存指令的冲突检测机制。冲突比较器结合数学模型可以准确判断访存地址范围的交集，能够迅速识别指令间的潜在冲突并调整调度策略。该方法从硬件角度出发，不仅支持指令的异步并发执行，还保证了指令操作的原子性及指令序列执行的正确性。

4 实验评估与讨论

4.1 实验环境

本实验使用 C++ 语言开发了一个访存冲突模拟器，其输入为标准格式的矩阵访存指令流。指令包含读写方向、起始地址、间隔步长、数据段长和段数目等信息。由于片外和片上的地址空间不重叠，模拟器仅关注片外部分的地址。模拟器构建了指令队列和冲突比较函数，实现了访存冲突的动态判定。该模拟器的片外存储模块基于 Ramulator 模拟器[15]实现，用于评估所提出的访存队列调度算法以及冲突判定机制在 DLP 访存场景下的有效性。Ramulator 是一个高精度、时钟级的存储系统模拟器，广泛用于计算体系结构和存储层次结构研究[16-17]。该平台能够精确模拟不同类型存储器（如 DDR 存储器、LPDDR 存储器、HBM 存储器等）在 DLP 中的访问行为，为访存优化方案的评估提供了可靠的实验环境。

目前的 DLP 尚未实现矩阵访存冲突检测，开发者因此必须依赖复杂的异步操作和同步屏障规避冲突。针对这一问题，本工作提出基线方法，采用 Banerjee-Wolfe 测试、集合交集法和 Power 测试进行冲突检测。Banerjee-Wolfe 测试作为数组依赖性分析中的经典方法，其基本思路是求解循环下标表达式的取值上下界，并判断常量是否可能落于该区间内，但该方法仅提供必要条件，难以充分判定跨步访存冲突。该方法本质上等同于研究动机中提出的区间覆盖法策略，即通过比较等式两侧循环变量的上下界来推断 2 个地址范围是否存在重叠。地址集合相交法通过逐元素比较 2 个地址集合的交集以实现准确的冲突检测，虽然不会误判但计算复杂度较高，为此本文在遍历过程中充分利用地址连续性以大幅降低对比次数。此外，数组依赖性分析中的 Power 测试[8-9]借助傅里叶-莫茨金变量消除法（Fourier-Motzkin elimination, FME），对循环下标表达式以及定义域范围所构成的不等式进行求解，并辅以额外数学约束，实现了精确的地址冲突检测。上述 3 种方法互为补充，共同构成本研究的对比基线。

本实验从 2 个方面开展对比分析。其一，关注访存带宽利用率（MBU），该指标量化实际内存传输速率与理论峰值带宽之比，是评估片外存储器数据传输效率的关键参数。其二，侧重于方法的判定开销，展示新方法在计算效率上的优异表现。双重指标评估不仅使实验结果更为全面，也为后续访存调度策略的优化提供了有价值的参考。为验证所提方法的优势，本文在访存带宽利用率方面将其与 Banerjee-Wolfe 测试进行对比，并探讨分块大小等参数对整体带宽的影响；在判定开销方面，则与 Power 测试及地址集合相交法比较，充分展示了所提方法在降低判定开销与提升系统效率方面的贡献。

实验平台的具体配置如下：存储系统基于 Ramulator 模拟片外存储器的读写，存储器使用当前寒武纪 MLU 配置的 LPDDR 存储器与 HBM 存储器；基准测试方面，实验选取线性层中的矩阵乘法（MM）、注意力机制中的批量矩阵乘法（BMM）、前馈神经网络中的 MLP 以及残差连接中的逐元素加法（Add）作为测试算子。这些算子在以大模型为代表的深度学习计算核心中占据重要位置。访存指令流基于寒武纪硬件架构与编程模型导出，以确保实验结果具有实际硬件指导意义。

4.2 实验结果

首先，我们以 Banerjee-Wolfe 测试作为基准对比不定方程求解带来的带宽利用率提升。图 7 展示了基于不定方程求解的细粒度判断和基于范围检测的粗粒度区间覆盖法的带宽利用率。实验结果表明：基于不定方程求解的细粒度判断，在 4 个测试例上，LPDDR 存储器的带宽利用率平均值从 72% 提升至 89%，HBM 存储器的带宽利用率平均值从 32% 提升至 88%，访存带宽利用率最高分别可达 94% 和 91%。

接着，我们讨论了不同拆分策略与数据分布对带宽利用率的影响。如图 8 所示，Config1~Config4 代表矩阵乘法计算时 4 种不同的分块大小（分别是

64×64, 64×128, 128×128, 256×128)。实验结果表明,无论开发者选择何种分块大小,最终的存储带宽利用率均可大幅提升。更一般地,在访存指令开放给开发者的情况下,如果指令之间存在冲突,本方法可确保指令执行的原子性;若指令之间无冲突,则可并发执行访存指令,充分利用存储通道,从而提升读写带宽,对于批量矩阵乘法、逐元素加法等访存密集型算子尤为关键。

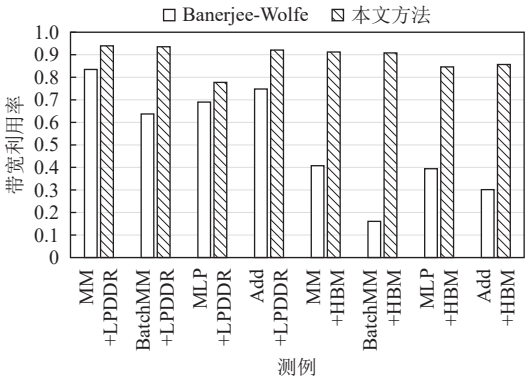


Fig. 7 Memory bandwidth utilization compared with Banerjee-Wolfe

图 7 与 Banerjee-Wolfe 对比访存带宽利用率

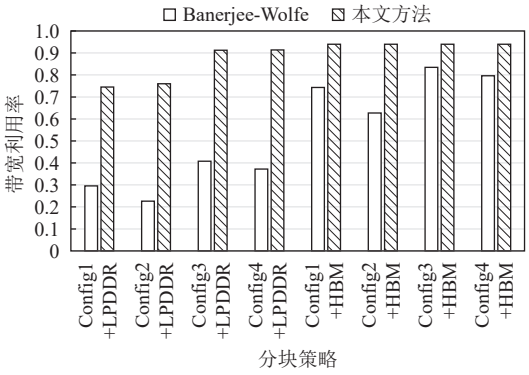


Fig. 8 Memory bandwidth utilization under different tiling strategies

图 8 不同分块策略下的访存带宽利用率

此外,本文探讨了内存通道数对访存带宽利用率的提升效应,并以高通道数 HBM 存储器为研究对象。在 8, 16, 32 通道配置下,本文以带宽利用率加速比作为指标,对比评估了基于 Banerjee-Wolfe 测试的优化效果。如图 9 所示,随着内存通道数由 8 提升至 32,对应的平均加速比分别为 1.348、1.744 和 2.964,表明本文方法在访存冲突检测与带宽利用率提升上具有显著优势。因此,当访存指令不存在数据冲突时,系统便能充分利用多通道内存,实际带宽更接近理论峰值;而现代 DLP 普遍采用多通道、高带宽的

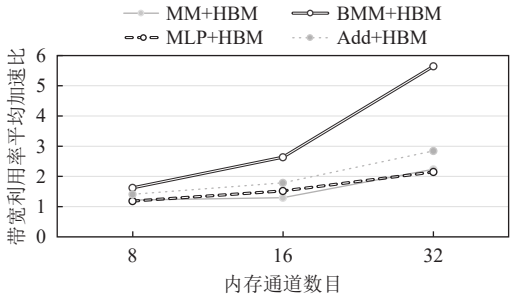


Fig. 9 Impact of memory channel numbers on bandwidth utilization ratios enhancement

图 9 内存通道数对带宽利用率提升的影响

HBM 存储架构(每个 stack 中通常含 8~16 个独立通道,整体配置 4~8 个 stack),为冲突判定机制提供了充分发挥的空间。

在执行效率方面,我们共设置了 8 组测试用例,其中 4 组采用之前的 MM, BMM, MLP, Add 测例,另外 4 组则通过在基准指令流中基于特定规则注入冲突,最终在每 1 万条指令中引入约 2 000 次冲突(约 20% 冲突),采用几何均值统计执行时间。如图 10 所示,实验结果表明,基于不定方程求解的细粒度判断方法,其平均执行时间仅为 Power 测试的 4.68%。与基于集合交集判断的方法相比,其平均执行时间仅为 0.31%,同时可以看出在注入冲突后的样例中执行时间加速效果下降。这种差异主要归因于:集合交集方法在无冲突时必须遍历所有相关元素(与矩阵数据量有关),而在检测到冲突时能利用逻辑短路提前退出;而本文方法利用扩展欧几里得算法求解不定方程始终保持较低的计算开销。实验结果充分证明了本文方法在计算复杂度和执行效率上的显著优势。

最后,在硬件实现方面,我们完成了比较器的设

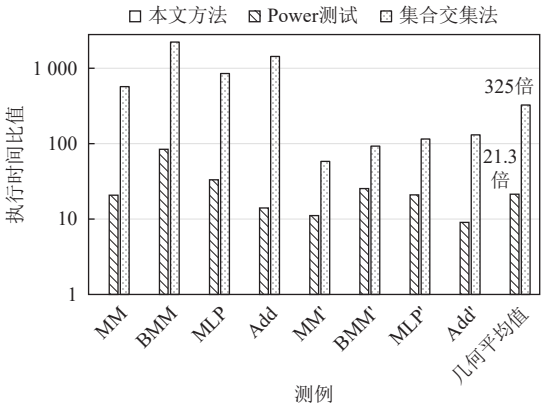


Fig. 10 Determination overhead compared with Power test and the method of set intersection

图 10 与 Power 测试和集合交集法对比判定开销

计、综合与布局，如图 11 所示，并重点关注比较器模块在面积、功耗与时序特性上的表现。需要说明的是，硬件实现中不对 DLP 中原本配置的访存队列进行评估，仅对本文方法新增部分进行分析。具体而言，比较器模块实现了算法 1 中的函数 *HasConflict*，并配置了 2 个 *Compare* 的计算单元以支持并行处理以及提升计算效率。基于 12 nm 工艺的综合结果显示，该比较器模块的总单元面积仅为 0.023 33 mm^2，这一数据远低于当前主流 DLP 芯片的面积规模（如英伟达 GA100 芯片面积达 826 mm^2），硬件开销上极其微小。同时，模块功耗仅为 4.119 4 mW，在满足系统性能要求的前提下，实现了低功耗运行。此外，尽管在计算过程中广泛采用了单周期除法运算，但模块的关键路径长度仅为 15.27 ns，表明数据传输延时较短。

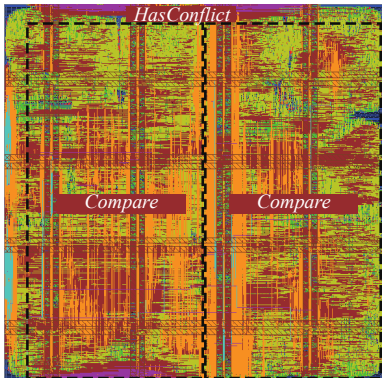


Fig. 11 Layout of comparator module

图 11 比较器模块布局版图

5 相关工作

DLP 是为了高效运行深度学习模型而设计的加速器。2015 年，寒武纪架构提出：DLP 应包含矩阵功能单元、向量功能单元和标量功能单元。其中，矩阵功能单元是处理核心，专为高效率的卷积和矩阵运算设计；其余任务由向量功能单元和标量功能单元协同完成。自国际上首个 DLP 架构 DianNao 及其扩展系列问世以来，加速深度学习成为领域内主要研究方向[18-21]，多种 DLP 在计算成本和效率方面取得了重要突破，数据流图简化了程序开发难度[22]，共同促进了卷积神经网络、图神经网络和大语言模型的发展[23-25]。

目前，寒武纪 MLU 思元处理器、华为 Ascend 昇腾处理器及英伟达 Hopper GPU 均支持基于矩阵粒度的跨步访存指令，从而显著提升片外存储与片上内存之间的数据传输效率。寒武纪依托高效的矩阵访存指令；华为 Ascend 处理器则采用模板化的切片数据搬运接口[13]；而英伟达 Hopper GPU 则通过 TMA 机制[14]，使用包含全局维度、步长及数据类型信息的张量映射描述符，使硬件直接完成复杂地址计算与数据传输。此外，对于寒武纪与华为 Ascend 处理器架构而言，即便在处理诸如 masked_select 等随机访存需求时，其数据加载阶段仍依赖于矩阵跨步访存，先执行固定步长的跨步数据搬运，再利用专用指令在片上完成随机选取，从而充分发挥规则跨步访存的高效优势。

1）数据竞争。数据竞争（data race）是指在并发程序中，2 个或多个线程在没有适当同步的情况下同时访问同一共享内存位置，并且至少有 1 个访问是写操作时发生的现象。这种情况下，线程间的访问顺序未被明确控制，可能导致非确定性行为，从而引发程序的不可预测错误或异常。数据竞争检测技术用于识别程序中潜在的数据竞争风险[26-27]，CPU 领域已有较为成熟的工具，而 GPU 环境下的相关研究仍在探索中[28-29]。矩阵访存指令的冲突判定机制在硬件层面减少了数据竞争风险，硬件通过提供原子性的矩阵访存操作，确保 1 次访存过程不可分割，避免非确定性行为。这一机制在一定程度上能够帮助开发者简化程序的同步逻辑，减少特定情况下的数据竞争发生概率，并间接降低数据竞争检测工具的工作负担。然而，矩阵访存指令的原子性无法完全避免数据竞争，开发者仍需关注深度学习负载的数据分块以及同步策略，以避免更高层次的竞争问题。

2）数组依赖性分析。数组依赖性分析是编译器优化中常用的静态分析技术，用于识别循环中数组访问的顺序，从而保证并行化转换的正确性。Banerjee 测试结合循环内下标表达式的取值范围来判断依赖关系，该方法通常较为保守。为了提高准确度，Power 测试采用傅里叶–莫茨金变量消除法，并引入额外数学约束，该方法能处理耦合下标和复杂循环边界，但计算成本较高。与此同时，Omega 测试利用整数线性规划将依赖判定转化为整数可行性检测，理论上在最坏情况下复杂度最高。近期的研究进展进一步聚焦于混合分析方法[30-31]，旨在结合编译时静态分析与运行时动态信息，以应对传统静态方法难以处理的复杂代码模式。此外，还可以集成指针别名分析[32-33]，准确定位内存访问，降低虚假依赖。

6 结 论

本文针对DLP中关键且高频的矩阵跨步访存模式，提出了一种准确且低开销的冲突检测方法。本研究在DLP架构设计中实现了一种矩阵跨步访存冲突判定机制，确保了访存指令的原子性，防止了未定义行为导致的不可预测性。该机制将矩阵访存地址比较归结为二元一次不定方程求解，通过形式化方法实现了精确且高效的数学建模。同时该机制可有效提升DLP在多通道存储器配置下的带宽利用率。此外，考虑到开放软件生态对硬件访存管理机制的依赖性，本文提出的冲突判定机制为开发者提供了明确的访存指令编程界面，增强了DLP计算生态的可用性和可靠性。

未来工作中将探索更安全高效的指令队列预取策略，更适合实际场景数据范围的除法实现方案，以进一步缩短关键路径延时，进一步提升系统效率。

致谢：感谢许健行、郑子夫和郭泓锐对本文内容的讨论与帮助。

作者贡献声明：张振兴设计并实现算法、撰写论文；赵永威对论文选题、算法设计、论文结构提供了指导意见；文渊博、郭崎和陈云霁提供了指导意见。

参 考 文 献

[1] Dhilleswararao P, Boppu S, Manikandan M S, et al. Efficient hardware architectures for accelerating deep neural networks: Survey[J]. IEEE Access, 2022, 10: 131788−131828

[2] Zhang Zhenxing, Wen Yuanbo, Lyu Hanqi, et al. AI computing systems for large language models training[J]. Journal of Computer Science and Technology, 2025, 40(1): 6−41

[3] Liu Shaoli, Du Zidong, Tao Jinhua, et al. Cambricon: An instruction set architecture for neural networks[C]//Proc of the 43rd Int Symp on Computer Architecture. New York: ACM, 2016: 393−405

[4] Jouppi N P, Yoon D H, Ashcraft M, et al. Ten lessons from three generations shaped Google's TPUv4i[C]//Proc of the 48th Int Symp on Computer Architecture. New York: ACM, 2021: 1−14

[5] Cambricon. BANG C/C++ programming guide[EB/OL]. 2025[2025-02-20]. https://www.cambricon.com/docs/sdk_1.15.0/cntoolkit_3.7.2/programming_guide_1.7.0/index.html

[6] Tillet P, Kung H T, Cox D. Triton: An intermediate language and compiler for tiled neural network computations[C]//Proc of the 3rd SIGPLAN Int Workshop on Machine Learning and Programming Languages. New York: ACM, 2019: 10−19

[7] Zhang Hongbing, Xing Mingjie, Wu Yanjun, et al. Compiler technologies in deep learning co-design: A survey[J]. Intelligent Computing, 2023, 2: 0040

[8] Banerjee U. Dependence Analysis for Supercomputing[M]. Berlin: Springer, 2013

[9] Kennedy K, Allen J R. Optimizing Compilers for Modern Architectures: A Dependence-based Approach[M]. San Francisco, CA, Morgan Kaufmann, 2001

[10] Li Rui, Xu Yufan, Sukumaran-Rajam A, et al. Analytical characterization and design space exploration for optimization of CNNs[C]//Proc of the 26th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2021: 928−942

[11] Chen Yunji. AI Computing Systems[M]. 2nd ed. Beijing: China Machine Press, 2024 (in Chinese)
(陈云霁. 智能计算系统[M]. 2版. 北京：机械工业出版社，2024)

[12] Cambricon. BANG C developer guide[EB/OL]. 2025[2025-02-20]. https://www.cambricon.com/docs/sdk_1.15.0/cntoolkit_3.7.2/cambricon_bang_c_4.7.2/index.html

[13] Ascend. Ascend C DataCopy[EB/OL]. 2025[2025-02-20]. https://www.hiascend.com/document/detail/zh/canncommercial/800/apiref/ascendcopapi/atlasascendc_api_07_0104.html

[14] NVIDIA. NVIDIA Hopper architecture in-depth[EB/OL]. 2025[2025-02-20]. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[15] Kim Y, Yang Weikun, Mutlu O. Ramulator: A fast and extensible DRAM simulator[J]. IEEE Computer Architecture Letters, 2015, 15(1): 45−49

[16] Zaourar L, Benazouz M, Mouhagir A, et al. Multilevel simulation-based co-design of next generation HPC microprocessors[C]//Proc of the Int Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems. Piscataway, NJ: IEEE, 2021: 18−29

[17] Yun Sungmin, Kim B, Park J, et al. GraNDe: Near-data processing architecture with adaptive matrix mapping for graph convolutional networks[J]. IEEE Computer Architecture Letters, 2022, 21(2): 45−48

[18] Liang Shengwen, Wang Ying, Liu Cheng, et al. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks[J]. IEEE Transactions on Computers, 2020, 70(9): 1511−1525

[19] Ye Wenhua, Zhou Xu, Zhou J, et al. Accelerating attention mechanism on FPGAs based on efficient reconfigurable systolic array[J]. ACM Transactions on Embedded Computing Systems, 2023, 22(6): 1−22

[20] Wu Ruohan, Zhu Xianyu, Chen Junshi, et al. SWattention: Designing fast and memory-efficient attention for a new Sunway supercomputer[J]. The Journal of Supercomputing, 2024, 80(10):

13657−13680

[21] Zhang Qing, Liu Cheng, Liu Bo, et al. Cross-layer optimization for fault-tolerant deep learning[J]. Journal of Computer Research and Development, 2024, 61(6): 1370−1387 (in Chinese)
(张青, 刘成, 刘波, 等. 容错深度学习加速器跨层优化[J]. 计算机研究与发展, 2024, 61(6): 1370−1387)

[22] Dou Yong, Wang Jialun, Su Huayou, et al. The impact of data flow computing thinking on the development of computer architecture[J]. SCIENTIA SINICA Informationis, 2020, 50(11): 1673−1713 (in Chinese)
(窦勇, 王嘉伦, 苏华友, 等. 从计算机体系结构发展历程看数据流计算思想[J]. 中国科学: 信息科学, 2020, 50(11): 1673−1713)

[23] Ge Xuran, Ou Yang, Wang Bo, et al. Survey of storage optimization techniques in large language model inference[J]. Journal of Computer Research and Development, 2025, 62(3): 545−562 (in Chinese)
(葛旭冉, 欧洋, 王博, 等. 大语言模型推理中的存储优化技术综述[J]. 计算机研究与发展, 2025, 62(3): 545−562)

[24] Shah J, Bikshandi G, Zhang Y, et al. Flashattention-3: Fast and accurate attention with asynchrony and low-precision[J]. Advances in Neural Information Processing Systems, 2024, 37: 68658−68685

[25] Liu Aixin, Feng Bei, Xue Bing, et al. DeepSeek-v3 technical report[J]. arXiv preprint, arXiv: 2412.19437, 2024

[26] Chen Le, Ding Xianzhong, Emani M, et al. Data race detection using large language models[C]//Proc of the Workshops of the Int Conf on High Performance Computing, Network, Storage, and Analysis. New York: ACM, 2023: 215−223

[27] Zheng Long, Li Yang, Xin Jie, et al. Minimal context-switching data race detection with dataflow tracking[J]. Journal of Computer Science and Technology, 2024, 39(1): 211−226

[28] Wu Mingyuan, Zhang Lingming, Liu Cong, et al. Automating CUDA synchronization via program transformation[C]//Proc of the 34th Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2019: 748−759

[29] Yu Guangba, Tan Gou, Huang Haojia, et al. A survey on failure analysis and fault injection in AI systems[J]. arXiv preprint, arXiv: 2407.00125, 2024

[30] Strout M M, Hall M, Olschanowsky C. The sparse polyhedral framework: Composing compiler-generated inspector-executor code[J]. Proceedings of the IEEE, 2018, 106(11): 1921−1934

[31] Apostolakis S, Xu Ziyang, Chan G, et al. Perspective: A sensible approach to speculative automatic parallelization[C]//Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 351−367

[32] Sui Yulei, Fan Xiaokang, Zhou Hao, et al. Loop-oriented pointer analysis for automatic simd vectorization[J]. ACM Transactions on Embedded Computing Systems, 2018, 17(2): 1−31

[33] Lepori A, Calotoiu A, Hoefler T. Iterating pointers: Enabling static analysis for loop-based pointers[J]. ACM Transactions on Architecture and Code Optimization, 2025, 22(1): 1−25



Zhang Zhenxing, born in 1996. PhD. His main research interests include computer architecture and parallel computing.
张振兴, 1996 年生。博士。主要研究方向为计算机体系结构、并行计算。



Zhao Yongwei, born in 1993. PhD, associate professor. Member of CCF. His main research interests include intelligent processors and computer architecture.
赵永威, 1993 年生。博士, 副研究员。CCF 会员。主要研究方向为智能处理器、计算机体系结构。



Wen Yuanbo, born in 1997. PhD, associate professor. Member of CCF. His main research interests include computer architecture and deep learning compiler.
文渊博, 1997 年生。博士, 副研究员。CCF 会员。主要研究方向为计算机体系结构、深度学习编译器。



Guo Qi, born in 1985. PhD, professor. Senior member of CCF. His main research interests include computer architecture, machine learning, and programming models.
郭　崎, 1985 年生。博士, 研究员。CCF 高级会员。主要研究方向为计算机体系结构、机器学习、编程模型。



Chen Yunji, born in 1983. PhD, professor. Fellow of CCF. His main research interests include parallel computing, microarchitecture, hardware verification, and computational intelligence.
陈云霁, 1983 年生。博士, 研究员。CCF 会士。主要研究方向为并行计算、微体系结构、硬件验证、计算智能。